Fast and Scalable Computation of the Forward and Inverse Discrete Periodic Radon Transform

Cesar Carranza, *Student Member, IEEE*, Daniel Llamocca, *Senior Member, IEEE*, and Marios Pattichis, *Senior Member, IEEE*

***Abstract*—The discrete periodic radon transform (DPRT) has extensively been used in applications that involve image reconstructions from projections. Beyond classic applications, the DPRT can also be used to compute fast convolutions that avoids the use of floating-point arithmetic associated with the use of the fast Fourier transform. Unfortunately, the use of the DPRT has been limited by the need to compute a large number of additions and the need for a large number of memory accesses. This paper introduces a fast and scalable approach for computing the forward and inverse DPRT that is based on the use of: 1) a parallel array of fixed-point adder trees; 2) circular shift registers to remove the need for accessing external memory components when selecting the input data for the adder trees; 3) an image block-based approach to DPRT computation that can fit the proposed architecture to available resources; and 4) fast transpositions that are computed in one or a few clock cycles that do not depend on the size of the input image. As a result, for an $N \times N$ image (N prime), the proposed approach can compute up to N^2 additions per clock cycle. Compared with the previous approaches, the scalable approach provides the fastest known implementations for different amounts of computational resources. For example, for a 251×251 image, for approximately 25% fewer flip-flops than required for a systolic implementation, we have that the scalable DPRT is computed 36 times faster. For the fastest case, we introduce optimized architectures that can compute the DPRT and its inverse in just $2N + \lceil \log_2 N \rceil + 1$ and $2N + 3\lceil \log_2 N \rceil + B + 2$ cycles, respectively, where B is the number of bits used to represent each input pixel. On the other hand, the scalable DPRT approach requires more 1-b additions than for the systolic implementation and provides a tradeoff between speed and additional 1-b additions. All of the proposed DPRT architectures were implemented in VHSIC Hardware Description Language (VHDL) and validated using an Field-Programmable Gate Array (FPGA) implementation.**

***Index Terms*—Scalable architecture, radon transform, parallel architecture, FPGA.**

Manuscript received February 7, 2015; revised July 8, 2015 and October 29, 2015; accepted November 12, 2015. Date of publication November 18, 2015; date of current version December 3, 2015. This work was supported by the National Science Foundation through the Division of Computer and Network Systems under Grant NSF AWD CNS-1422031. The associate editor coordinating the review of this manuscript and approving it for publication was Dr. Paul Rodriguez.

C. Carranza is with the Department of Electrical and Computer Engineering, University of New Mexico, Albuquerque, NM 87131 USA, and also with the Engineering Department, Pontificia Universidad Católica del Perú, Lima 32, Peru (e-mail: acarran@pucp.edu.pe).

D. Llamocca is with the Electrical and Computer Engineering Department, Oakland University, Rochester, MI 48309 USA (e-mail: llamocca@oakland.edu).

M. Pattichis is with the Department of Electrical and Computer Engineering, University of New Mexico, Albuquerque, NM 87131 USA (e-mail: pattichi@unm.edu).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TIP.2015.2501725

I. Introduction

THE DISCRETE Radon Transform (DRT) is an essential component of a wide range of applications in image processing [1], [2]. Applications of the DRT include the classic application of reconstructing objects from projections in computed tomography, radar imaging, and magnetic resonance imaging [1], [2]. Furthermore, the DRT has also been applied in image denoising [3], image restoration [4], texture analysis [5], line detection in images [6], and encryption [7]. More recently, the DRT has been applied in erasure coding in wireless communications [8], signal content delivery [9], and compressive sensing [10].

A popular method for computing the DRT involves the use of the Fast Fourier Transform (FFT). The basic approach is to sample the 2D FFT along different radial lines through the origin and then use the 1D inverse FFT along each line to estimate the DRT. This direct approach suffers from many artifacts that have been discussed in [3]. Assuming that the DRT is computed directly, Beylkin proposed an exact inversion algorithm in [11]. A significant improvement to this approach was proposed by Kelley and Madisetti by eliminating interpolation calculations [12]. A common way to address this complexity is to use Graphic Processing Unit (GPU) implementations as described in [13]. Unfortunately, this earlier work on the DRT requires the use of expensive floating point units for implementing the FFTs. Floating point units require significantly larger amounts of hardware resources than fixed point implementations that will be discussed next.

Fixed point implementations of the DRT can be based on the Discrete Periodic Radon Transform (DPRT). Grigoryan first introduced the forward DPRT algorithm for computing the 2D Discrete Fourier Transform as discussed in [14]. In related work, Matus and Flusser presented a model for the DPRT and proposed a sequential algorithm for computing the DPRT and its inverse for prime sized images [15]. This research was extended by Hsung et al. for images of sizes that are powers of two [16].

Similar to the continuous-space Radon Transform, the DPRT satisfies discrete and periodic versions of the the Fourier slice theorem and the convolution property. Thus, the DPRT can lead to efficient, fixed-point arithmetic methods for computing circular and linear convolutions as discussed in [16]. The discrete version of the Fourier slice theorem provides a method for computing 2D Discrete Fourier Transforms based on the DPRT and a minimal number of 1D FFTs (e.g., [14], [17]).

A summary of DPRT architectures based on the algorithm described by [15] can be found in [18]. In [15], the DPRT of an image of size $N \times N$ (N prime) requires $(N+1)N(N-1)$ additions. Based on the algorithm given in [15], a serial and power efficient architecture was proposed in [19]. In [19], the authors used an address generator to generate the pixels to add. The DPRT sums were computed using an accumulator adder that stores results from each projection using N shift registers. The serial architecture described in [19] required resources that grow linearly with the size of the image while requiring $N(N^2 + 2N + 1)$ clock cycles to compute the full DPRT.

Also based on the algorithm given in [15], a systolic architecture implementation was proposed in [20]. The architecture used a systolic array of $N(N + 1)(\log_2 N)$ bits to store the addresses of the values to add. The pixels are added using using $(N + 1)$ loop adder blocks. The data I/O was handled by $N+1$ dual-port RAMs. For this architecture, resource usage grows as $O(N^2)$ at a reduced running time of $N^2 + N + 1$ cycles for the full DPRT.

The motivation for the current manuscript is to investigate the development of DPRT algorithms that are both fast and scalable. Here, we use the term *fast* to refer to the requirement that the computation will provide the result in the minimum number of cycles. Also, we use the term *scalable* to refer to the requirement that the approach will provide the fastest implementation based on the amounts of available resources.

This manuscript is focused on the case that the image is of size $N \times N$ and N is prime. For prime N, the DPRT provides the most efficient implementations by requiring the minimal number of $N+1$ primal directions [21]. In contrast, there are $3N/2$ primal directions in the case that $N = 2^p$ where p is a positive integer [22]. On the other hand, despite the additional directions, it is possible to compute the directional sums faster for $N = 2^p$, as discussed in [23] and [24]. However, it is important to note that prime-numbered transforms have advantages in convolution applications. Here, just like for the Fast Fourier Transform (FFT), we can use zero-padding to extend the DPRT for computing convolutions in the transform domain. Unfortunately, when using the FFT with $N = 2^p$, zero-padding requires that we use FFTs with double the size of N. In this case, it is easy to see that the use of prime-numbered DPRTs is better since there are typically many prime numbers between 2^p and 2^{p+1}. For example, it can be shown that the n-th prime number is approximately $n \log(n)$ which gives an approximate sequence of primes that are $n \log(n), (n + 1) \log(n + 1)$ which is a lot more dense than what we can accomplish with powers of two $2^n, 2^{n+1}$ [25]. As a numerical example, there are 168 primes that are less than 1000 as opposed to just 9 powers of 2. Thus, instead of doubling the size of the transform, we can use a DPRT with only a slightly larger transform.

This manuscript introduces a fast and scalable approach for computing the forward and inverse DPRT that is based on parallel shift and add operations. Preliminary results were presented in conference publications in [26] and [27]. The conference paper implementations were focused on special cases of the full system discussed here, required an external system to add the partial sums, assumed pre-existing hardware for transpositions, and worked with image strip-sizes that were limited to powers of two. The current manuscript includes: (i) a comprehensive presentation of the theory and algorithms, (ii) extensive validation that does not require external hardware for partial sums and transpositions, (iii) works with arbitrary image strip sizes, and also includes (iv) the inverse DPRT. In terms of the general theory presented in the current manuscript, the conference paper publications represented some special cases. The contributions of the current manuscript over previously proposed approaches are summarized in the following paragraphs.

Overall, a fundamental contribution of the manuscript is that it provides a fast and scalable architecture that can be adapted to available resources. Our approach is designed to be fast in the sense that column sums are computed on every clock cycle. In the fastest implementation, a prime direction of the DPRT is computed on every clock cycle. More generally, our approach is scalable, allowing us to handle larger images with limited computational resources.

Furthermore, the manuscript provides a Pareto-optimal DPRT and inverse DPRT based on running time and resources measured in terms of one-bit additions (or 1-bit full-adders) and flip-flops. Thus, the proposed approach is shown to be Pareto-optimal in terms of the required cycles and required resources. Here, Pareto-optimality refers to solutions that are optimal in a multi-objective sense (e.g., see [28]). Thus, in the current application, Pareto-optimality refers to the fact that the scalable approach provides the fastest known implementations for the given computational resources. As an example, in the fastest case, for an $N \times N$ image (N prime), we compute the DPRT in linear time ($2N + \lceil \log_2 N \rceil + 1$ clock cycles) requiring resources that grow quadratically ($O(N^2)$). In the most limited resources case, the running time is quadratic ($\lceil N/2 \rceil (N+9) + N + 2$ clock cycles) requiring resources that grow linearly ($O(N)$). A Pareto-front of optimal solutions is given for resources that fall within these two extreme cases. All prior research in this area focused on the development of a single architecture. We also obtained similar results for the inverse DPRT, although results for this case were not previously reported.

In terms of speed, the manuscript describes the fastest possible implementation of the DPRT and inverse DPRT. For the fastest cases, assuming sufficient resources for implementation, we introduce the fast DPRT (FDPRT) and the fast inverse DPRT (iFDPRT) that can compute the full transforms in $2N + \lceil \log_2 N \rceil + 1$ and $2N + 3 \lceil \log_2 N \rceil + B + 2$ cycles respectively (B is the number of bits used to represent each input pixel).

To achieve the performance claims, we describe a parallel and pipelined implementation that provides an improvement over the sequential algorithm proposed by [15] and used in [19] and [20]. To summarize the performance claims, let the $N \times N$ input image be sub-divided into strips of H rows of pixels. Then, for $H = 2, \ldots, (N - 1)/2$, our scalable approach computes $N \times H$ additions in a single clock cycle. Furthermore, shift registers are used to make data available to the adders in every clock cycle. Then, additions and shifts are performed in parallel in the same clock cycle.

In addition, we implement the use of fast transpositions. We propose two unique transpositions methods. First, we have a RAM-based architecture and associated algorithm that provides a complete row or column of the input image in one clock cycle. Using this parallel RAM access architecture, transposition is avoided since the image can be accessed by either rows or columns. Second, we use a register-based architecture that transpose the complete register array in one clock cycle. This second approach avoids the use of RAMs.

Finally, we provide a generic and parametrized family of architectures that is validated with FPGA implementations. Thus, the proposed architectures are not tied to any particular hardware. They can be applied to any existing hardware (e.g., FPGA or VLSI) since they were developed in VHDL and are fully parametrized for any prime N.

The rest of the manuscript is organized as follows. The mathematical definitions for the DPRT and its inverse are given in section II. The proposed approach is given in section III. Section IV describes the architecture implementation on a FPGA. Section V presents the results. Conclusions and future work are given in section VI.

II. Background

The purpose of this section is to introduce the basic definitions associated with the DPRT and provide a very brief summary of previous implementations. We introduce the notation in section II-A. We then produce the definitions of the DPRT and its inverse in section II-B. A summary of previous implementations is given in section II-C.

A. Notation Summary

We begin by introducing the notation. We consider $N \times N$ images where N is prime. We let Z_N denote the non-negative integers: $\{0, 1, 2, \ldots, N-1\}$, and $l^2(Z_N^2)$ be the set of square-summable functions over Z_N^2. Then, let $f \in l^2(Z_N^2)$ be a 2D discrete function that represents an $N \times N$ image, where each pixel is a positive integer value represented with B bits. Also, we use subscripts to represent rows. For example, $f_k(j)$ denotes the vector that consists of the elements of f where the value of k is fixed. Similarly, for $R(r, m, d)$, $R_{r,m}(d)$ denotes the vector that consists of the elements of R with fixed values for r, m. Here, we note that we are always fixing all but the last index. We use $\langle \alpha \rangle_\beta$ to denote the modulo function. In other words, $\langle \alpha \rangle_\beta$ denotes the positive remainder when we divide α by β where $\alpha, \beta > 0$.

To establish the notation, we consider an example. For an 251×251 8-bit image, we have $N = 251$, $B = 8$, and f represents the image. We then have that $f_1(j)$ represents the first row in the image. In 3-dimensions, $R_{1,2}(d)$ denotes the elements $R(r = 1, m = 2, d)$, where d id allowed to vary. For the modulo-notation, we have $\langle 255 \rangle_{251} = 4$ which represents the integral remainder when we divde 255 by $N = 251$. We use $R(m, d)$ to denote the DPRT of f and $R'(r, m, d)$ to index the r-th partial sum associated with $R(m, d)$. Here, R' is used for explaining the computations associated with the scalable DPRT.

B. Discrete Periodic Radon Transform and Its Inverse

We introduce the definition of the DPRT and its inverse (iDPRT) based on [16]. Let f be square-summable. The DPRT of f is also square summable and given by:

$$R(m, d) = \begin{cases} \sum_{i=0}^{N-1} f(i, \langle d + mi \rangle_N), & 0 \le m < N, \\ \sum_{j=0}^{N-1} f(d, j), & m = N, \end{cases} \tag{1}$$

where $d \in Z_N$ and $m \in Z_{N+1}$. In (1), we observe that m is used to index the prime directions as documented in [26].

The iDPRT recovers the input image as given by:

$$f(i, j) = \frac{1}{N} \left[\sum_{m=0}^{N-1} R\left(m, \langle j - mi \rangle_N\right) - S + R(N, i) \right] \tag{2}$$

where:

$$S = \sum_{j=0}^{N-1} \sum_{i=0}^{N-1} f(i, j). \tag{3}$$

From (3), it is clear that S represents the sum of all of the pixels. Since each projection computes the sums over a single direction, we can sum up the results from any one of these directions to compute S as given by:

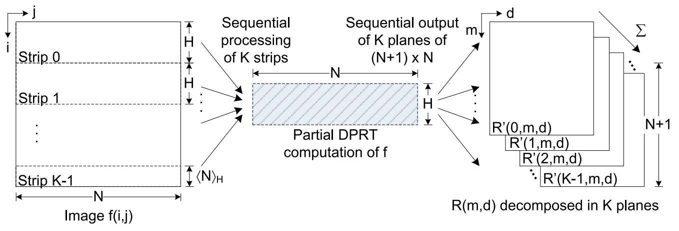
$$S = \sum_{d=0}^{N-1} R(m, d). \tag{4}$$

We note that the DPRT as given by (1) requires the computation of $N + 1$ projections. All of these projections are used in the computation of iDPRT as given in (2). In (2), the last projection computes $R(N, i)$ that is needed in the summation.

C. DPRT Implementations

DPRT implementations have focused on implementing the algorithm proposed in [15]. The basic algorithm is sequential that relies on computing the indices i, j to access $f(i, j)$ that are needed for the additions in (1). For each prime direction, as shown in (1), the basic implementation requires N^2 memory accesses and $N(N-1)$ additions. For computing all of the prime directions $(N + 1)$, we thus have $(N + 1)N^2$ memory accesses and $(N + 1)N(N - 1)$ additions.

Based on [15], hardware implementations have focused on computing memory indices, followed by the necessary additions [19], [20]. An advantage of the serial architecture given in [19] is that it requires hardware resources that grow linearly with N (for and $N \times N$ image). Unfortunately, this serial architecture leads to slow computation since it computes the DPRT in a cubic number of cycles ($N(N^2 + 2N + 1)$ clock cycles). A much faster, systolic array implementation was presented in [20]. The systolic array implementation computes N indices and N additions per cycle. Overall, the systolic array implementation requires hardware resources that grow quadratically with N while requiring $N^2 + N + 1$ clock cycles to compute the full DPRT.



j
i
Strip 0
Strip 1
Strip K-1
H
H
⟨N⟩H
N
Image f(i,j)
Sequential processing of K strips
Sequential output of K planes of (N+1) x N
N
H
Partial DPRT computation of f
d
m
Σ
R'(0,m,d)
R'(1,m,d)
R'(2,m,d)
R'(K-1,m,d)
N+1
R(m,d) decomposed in K planes

Fig. 1. Scalable DPRT concept. The input image is divided into K strips. The DPRT is computed by accumulating the partial sums from each strip.

The proposed architecture does not require memory indexing and computes the additions in parallel. Furthermore, the new architecture is scalable, and thus allows us a consider a family of very efficient architectures that can also be implemented with limited resources.

III. Methodology

This section presents a new fast algorithm and associated scalable architecture that can be used to control the running time and hardware resources required for the computation of the DPRT. Additionally, we extend the approach to the inverse DPRT (iDPRT). At the end of the section, we provide an optimized architecture implementation that computes the DPRT and iDPRT in the least number of clock cycles.

A. Partial DPRT

For the development of scalable architecture implementations, we introduce the concept of the partial DPRT. The basic concept is demonstrated in Fig. 1 and formally defined below.

The idea is to divide f into strips that contain H rows of pixels except for the last one that is composed of the remaining number of rows needed to cover all of the N rows (see Fig. 1). Here, we note that the height of the last strip will be $\langle N\rangle_H \neq 0$ since N is prime. Now, if we let K be the number of strips, we have that $K = \lceil N/H\rceil$. In what follows, let r denote the r-th strip. We compute the DPRT over each strip using:

$$R(m,d) = \begin{cases} \sum_{r=0}^{K-1}\sum_{i=0}^{L(r)-1} f(i+rH, \langle d+m(i+rH)\rangle_N), & \\ & 0 \le m < N \\ \sum_{r=0}^{K-1}\sum_{j=0}^{L-1} f(d, j+rH), & m = N \end{cases} \tag{5}$$

where

$$L(r) = \begin{cases} H, & r < K-1 \\ \langle N\rangle_H & r = K-1. \end{cases} \tag{6}$$

We let $R'(r,m,d)$ denote the r-th partial DPRT defined by:

$$R'(r,m,d) = \begin{cases} \sum_{i=0}^{L(r)-1} f(i+rH, \langle d+m(i+rH)\rangle_N), & \\ & 0 \le m < N \\ \sum_{j=0}^{L(r)-1} f(d, j+rH), & m = N \end{cases} \tag{7}$$

where, $r = 0, \ldots, K-1$ is the strip number. Therefore, the DPRT can be computed as a summation of partial DPRTs using:

$$R(m,d) = \sum_{r=0}^{K-1} R'(r,m,d). \tag{8}$$

Similarly, we define the partial iDPRT of $R(m,d)$ using

$$f'(r,i,j) = \sum_{m=0}^{L(r)-1} R(m+rH, \langle j - i(m+rH)\rangle_N) \tag{9}$$

which allows us to compute the iDPRT of $R(m,d)$ using a summation of partial iDPRTs:

$$f(i,j) = \frac{1}{N}\left[\sum_{r=0}^{K-1} f'(r,i,j) - S + R(N,i)\right]. \tag{10}$$

In what follows, we let let $n = \lceil \log_2 N\rceil$, $h = \lceil \log_2 H\rceil$, and $R'_{r,m}(d)$ be an N-th dimensional vector representing the partial DPRT of strip r.

B. Scalable Fast Discrete Periodic Radon Transform (SFDPRT)

In this section, we develop the scalable DPRT hardware architecture by implementing the partial DPRT concepts presented in Fig. 1. We present a top-level view of the hardware architecture for the scalable DPRT in Fig. 2 and the associated algorithm in Fig. 3. We refer to Fig. 1 for the basic concepts. The basic idea is to achieve scalability by controlling the number of rows used in each rectangular strip. Thus, for the fastest performance, we choose the largest pareto-optimal strip size that can be implemented using available hardware resources. The final result is computed by combining the DPRTs as given in (7).

We begin with an overview of the architecture as presented in Fig. 2. We have three basic hardware blocks: the input memory block (`MEM_IN`), the partial DPRT computation block (`SFDPRT_core`), and output/accumulator memory block (`MEM_OUT`). The input image f is loaded into the input buffer `MEM_IN` which can be implemented using a customized RAM that supports access to each image row or column in a single clock cycle. Partial DPRT computation is performed using the `SFDPRT_core`. We implement `SFDPRT_core` using an $H \times N$ register array with B bits depth so as to be able to store the contents of a single strip. Each row of the `SFDPRT_core` register array is implemented using a Circular Left Shift (CLS) register that can be used to align the image samples along each column. Each column of this array has a H-operand fully pipelined adder tree capable to add the complete column in one clock cycle. The output of the adder trees provide the output of the `SFDPRT_core`, which represents the partial DPRT of f. This combination of shift registers and adders allows the computation of $H \times N$ additions per clock cycle with a latency of h. At the end, the outputs of the `SFDPRT_core` are accumulated using `MEM_OUT`. We summarize the required computational resources in section V.

A fast algorithm for computing the DPRT is summarized in Fig. 3. We also present a detailed timing diagram for each

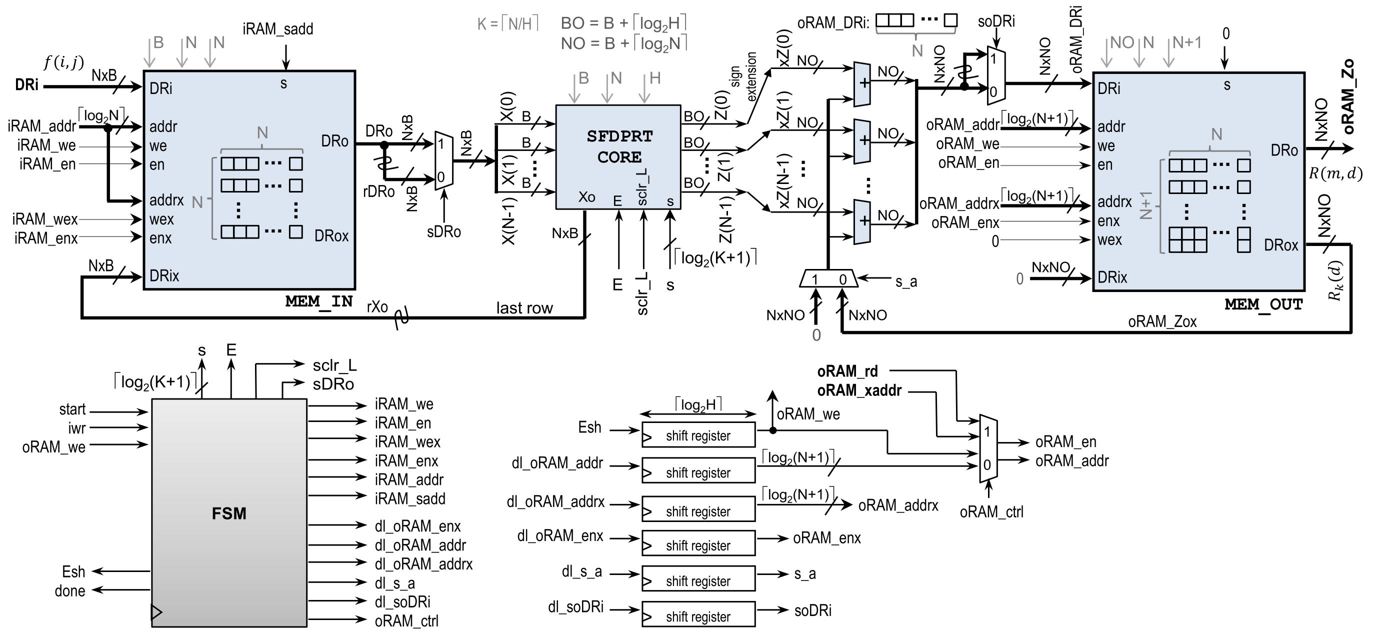


Fig. 2. System for implementing the Scalable and Fast DPRT (SFDPRT). The `SFDPRT_core` computes the partial sums. `MEM_IN` and `MEM_OUT` are dual port input and output memories. A Finite State Machine (`FSM`) is used for control. See text in Sec. III-B for more details.

```
 1: Load_shifted_image (f) in MEM_IN
        using CLS registers of SFDPRT_core.
 2: for $r$ = 0 to $K - 1$ do
 3:     Load_strip($r$,'row_mode') into the SFDPRT_core
 4:     for $k$ = 0 to $N - 1$ do
 5:         Shift in parallel all the $H$ rows:
                $CLS_a(H \cdot r + a)$, $a = 0, \ldots, H - 1$
 6:         Compute in parallel $R'_{r,k}(d)$
 7:         Add_partial_result: $R_k(d) = R_k(d) + R'_{r,k}(d)$
                in MEM_OUT
 8:     end for
 9: end for
10: for $r$ = 0 to $K - 1$ do
11:     Load_strip ($r$,'column_mode') into the SFDPRT_core
12:     Compute in parallel $R'_{r,N}(d)$
13:     Add_partial_result: $R_N(d) = R_N(d) + R'_{r,N}(d)$
            in MEM_OUT
14: end for
```

Fig. 3. Top level algorithm for computing the scalable and fast DPRT (SFDPRT). Within each loop, all of the operations are pipelined. Then, each iteration takes a single cycle. For example, the Shift, pipelined Compute, and the Add operations of lines 5, 6, and 7 are always computed within a single clock cycle. We refer to section II-A for the notation.

of the steps in Fig. 4. For the timing diagram, we note that time increases to the right. Along the columns, we label each step and the required number of cycles. Furthermore, computations that occur in parallel will appear along the same column. To understand the timing for each computation, recall that N denotes the number of image rows, K denotes the number of image strips where each strip contains a maximum of H image rows.

Furthermore, to explain the reduced timing requirements, we note the special characteristics of the pipeline structure. First, we use dual port RAMs (`MEM_IN` and `MEM_OUT`) that allow us to load and extract one image row per cycle. Thus, we start computing the first projection while we are still shifting (also see the overlap between the second and third computing steps of Fig. 4). Second, we note that we are using fully pipelined adder trees which allow us to start the computation of the next projection without requiring the completion of the previous projection (see overlap in projection computations in Fig. 4).

We next summarize the entire process depicted in Figs. 3 and 4. Initially, we load a shifted version of the image into `MEM_IN`. The significance of this step is that the stored image allows computation of the last projection in a single cycle without the need for transposition. Here, we note that we can access rows and columns of `MEM_IN` in a single clock cycle. In terms of timing, the process of loading and shifting in the image requires $N + K(H + 1)$ cycles.

Then, we compute the first N projections by loading each one of the K strips (outer loop) and then adding the partial results (inner loop). The partial DPRT for the strip r is computed in the inner loop, (see lines 4-8 in Fig. 3). For computing the full DPRT, the partial DPRT outputs are accumulated in `MEM_OUT`. In terms of timing, each strip requires $N + H + 1$ cycles as detailed in Fig. 4. Thus, it takes a total of $K(N + H + 1)$ cycles for computing the first N projections.

For the last projection, we note the requirement for special handling (see lines 10-14 in Fig. 3). This special treatment is due to the fact that unlike the first N projections that can be implemented effectively using shift and add operations of the rows, the last projection requires shift and add operations of the columns. For this last projection, we require $K(H + 1) + h+1$ cycles which brings the total to $K(N+3H+3)+N+h+1$ cycles for computing the full DPRT. Furthermore, the DPRT

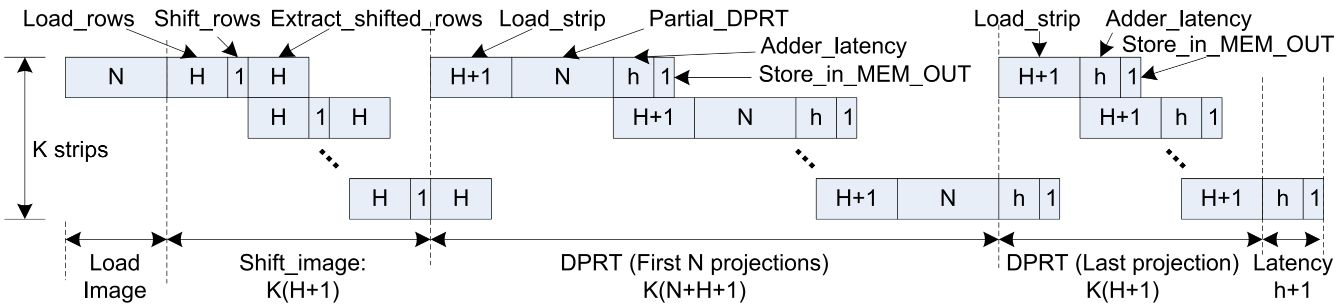


Fig. 4. Running time for scalable and fast DPRT (SFDPRT). In this diagram, time increases to the right. The image is decomposed into K strips. Then, the first strip appears in the top row and the last strip appears in the last row of the diagram. Here, H denotes the maximum number of image rows in each strip, $K = \lceil N/H \rceil$ is the number of strips, and $h = \lceil \log_2 H \rceil$ represents the addition latency.

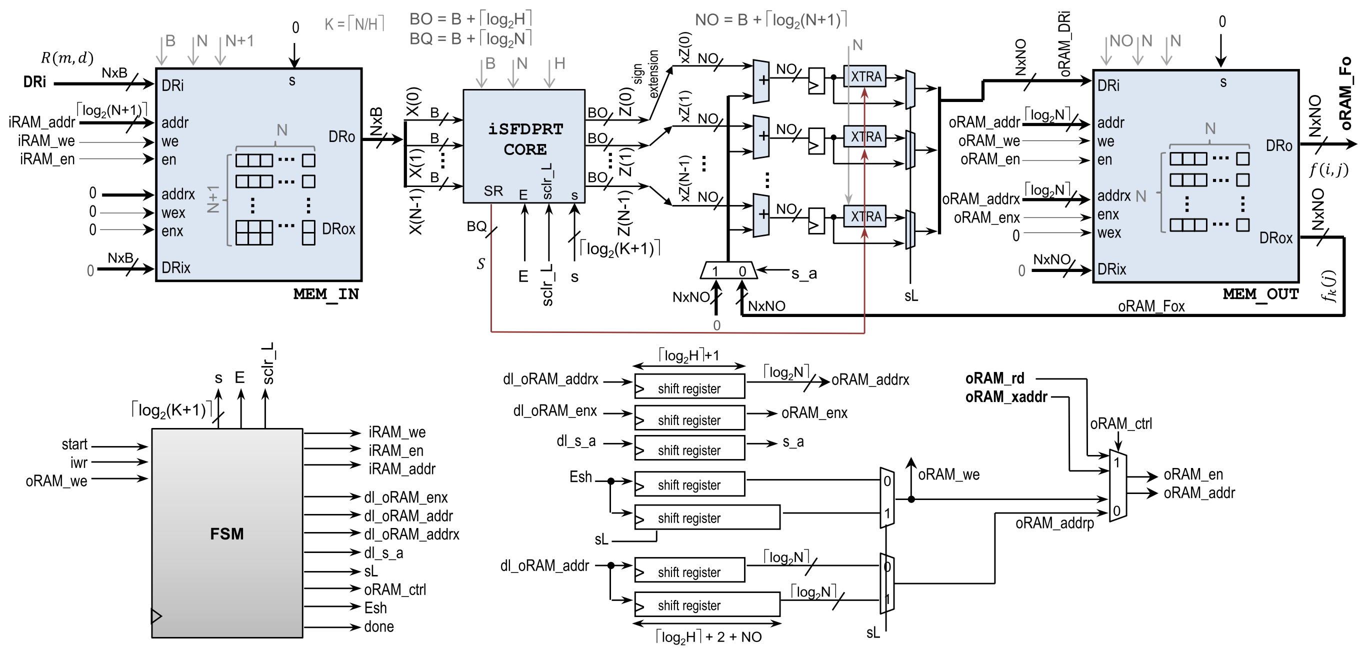


Fig. 5. System for implementing the inverse, scalable and fast DPRT (iSFDPRT). The system uses the `iSFDPRT_core` core for computing partial sums. The system uses dual port input and output memories, an accumulator array and a Finite State Machine for control. See text in Sec. III-C for more details.

is represented exactly by using $B + \lceil \log_2 N \rceil$ bits where B represents the number of input bits.

C. Inverse Scalable Fast Discrete Periodic Radon Transform (iSFDPRT)

The scalable architecture for the iDPRT is given in Fig 5, and the associated algorithm is given in Fig. 6. Here, we have three basic hardware blocks: (i) the input memory block (`MEM_IN`), (ii) the partial inverse DPRT computation block (`iSFDPRT_core`), and (iii) the output/accumulator memory block (`MEM_OUT`). The functionality of this system is the same as the SFDPRT (see Sec. III-B) with the exception of the `XTRA` circuit that performs the normalization of the output. Since there are many similarities between the DPRT and its inverse, we only focus on explaining the most significant differences. The list of the most significant differences include:

- **Input size:** The input is $R(m, d)$ with a size of $(N + 1) \times N$ pixels.
- **No transposition and optional use of `MEM_IN`:** A comparison between (1) and (2) shows that second term of (1) is not needed for computing (2). Thus, the horizontal sums that required fast transposition are no longer needed. As a result, `MEM_IN` is only needed to buffer/sync the incoming data. In specific implementations, `MEM_IN` may be removed provided that the data can be input to the hardware in strips as described in Fig. 6.
- **Circular right shifting replaces circular left shifts:** A comparison between (1) and (2) shows that the iDPRT index requires $\langle j - mi \rangle_N$ as opposed to $\langle d + mi \rangle_N$ for the DPRT. As a result, we have that the circular left shifts (CLS) of the DPRT become circular right shifts (CRS) for the iDPRT.

In terms of minor differences, we also note the special iDPRT terms of $R_N(d)$ and S in (2) that are missing from the DPRT. These terms needed to added (for $R_N(d)$) and subtracted (for S) for each summation term. Refer to Fig. 6 for details.

We consider an optimized implementation that uses pipelined dividers with a latency of as many clock cycles as the number of bits needed to represent the dividend.

```
1:  for $r$ = 0 to $K-2$ do
2:      Load strip $r$ into the iSFDPRT_core
3:      if $r = 0$ then
4:          Compute $S$
5:      end if
6:      for $k$ = 0 to $N-1$ do
7:          Shift in parallel all the $H$ rows:
                $CRS_a(H \cdot r + a)$,
                $a = 0, \dots, H-1$
8:          Compute in parallel $f'_{r,k}(j)$
9:          Add partial result $f_k(j) = f_k(j) + R'_{r,k}(j)$
                in MEM_OUT
10:     end for
11: end for
12: Load last strip into the iSFDPRT_core
13: for $k$ = 0 to $N-1$ do
14:     Shift in parallel $\langle N \rangle_H$ rows:
            $CRS_a(H \cdot r + a)$,
            $a = 0, \dots, \langle N \rangle_H - 1$
15:     Compute in parallel $f'_{r,k}(j) + R_N(d)$
16:     Add partial result: $f_k(j) = f_k(j) + f'_{r,k}(j) + R_N(d)$
17:     Subtract S: $f_k(j) = f_k(j) - S$
18:     Normalize by N: $f_k(j) = f_k(j)/N$
            and store in MEM_OUT
19: end for
```

Fig. 6. Top level algorithm for computing the inverse Scalable Fast Discrete Periodic Radon Transform $f(i, j) = \Re^{-1}(R(m, d))$. With the exception of the strip operations of lines 2 and 12, all other operations are pipelined and executed in a single clock cycle. The strip operations require H clock cycles where H represents the number of rows in the strip. We refer to section II-A for the notation.

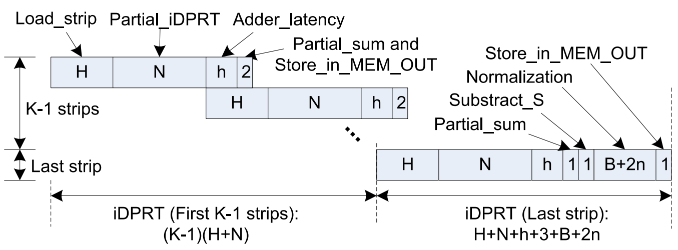


Fig. 7. Running time for computing the inverse, scalable, fast DPRT (iSFDPRT). Here, H denotes the maximum number of projection rows for each strip, $K = \lceil N/H \rceil$ is the number of strips, $h = \lceil \log_2 H \rceil$ is the addition latency, $n = \lceil \log_2 N \rceil$, and $B + 2n$ is the number of bits used to represent the results before normalization.

Then, the total running time is $K(N + H) + h + 3 + B + 2n$ as illustrated in Fig. 7. Resource requirements are given in section V.

D. Fast Discrete Periodic Radon Transform (FDPRT) and Its Inverse (iFDPRT)

When there are sufficient resources to work with the entire image, there is no need to break the image into strips. All the computations can be done in place without the need to compute partial sums that will later have to be accumulated. In this case, we eliminate the use of the RAM and simply hold the input in the register array. For this case, we use the terms FDPRT and iFDPRT to describe the optimized implementations. For the FDPRT, the register array is also modified to implement the fast transposition that is required for the last projection (transposition time=1 clock cycle).

We present an example of the FDPRT hardware implementation for an 7×7 image in Fig. 8. We also present the associated timing diagram in Fig. 9. Here, we note that time increases to the right. As before, the different computational steps are depicted along the columns. Cycles associated with parallel computations appear within the same column.

We next explain the process and derive the total running time in terms of the required number of cycles. Initially, the image is loaded row-by-row at the top as shown in Fig. 8. Thus, image loading requires N cycles as depicted in the timing diagram of Fig. 9. Shifting is performed in a single cycle along each row. The shifted rows are then added along each column as shown in Fig. 8. Due to the fully pipelined architecture, it only takes $N - 1$ cycles to compute the first $N - 1$ projections. The last two projections only require two additional cycles. Then, the final result is is only delayed by the latency associated with the last addition ($n = \lceil \log_2 N \rceil$). Thus, overall, it only takes $2N + n + 1$ cycles to compute the FDPRT.

For the iFDPRT, the architecture is basically reduced to the CRS registers plus the adder trees. For the iFDPRT, we also have an additional CLS(1), the subtraction of S and the normalizing factor ($1/N$) that is be embedded inside the adder trees. The iFDPRT algorithm is given in Fig. 10. Overall, the iFDPRT requires $2N + 3n + B + 2$ cycles. We summarize the required resources for both FDPRT and iFDPRT in section V.

E. Pareto-Optimal Realizations

For the development of scalable architectures, we want to restrict our attention to implementations that are optimal in the multi-objective sense. A similar approach was also considered in [29].

Basically, the idea is to expect that architectures with more hardware resources will also provide better performance. Here, we want to consider architectures that will give faster running times as we increase the hardware resources.

The set of implementations that are optimal in the multi-objective sense forms the Pareto front [28]. Formally, an implementation is considered to be sub-optimal if we can find another (different) implementation that can run at the same time or faster for the same or less hardware resources, excluding the case where both the running time and computational resources are equal. The Pareto front is then defined by the set of realizations that cannot be shown to be sub-optimal.

For deriving the Pareto-front, we fix the image size to N. Then, we want to find the number of rows in each image strip (values of H) that generate Pareto-optimal architectures. Now, since N is prime, it cannot be divided by H exactly. The number of strips is given by $\lceil N/H \rceil$ which denotes the ceiling function applied to N/H. To derive the Pareto-front, we require that larger values of H will result in fewer strips

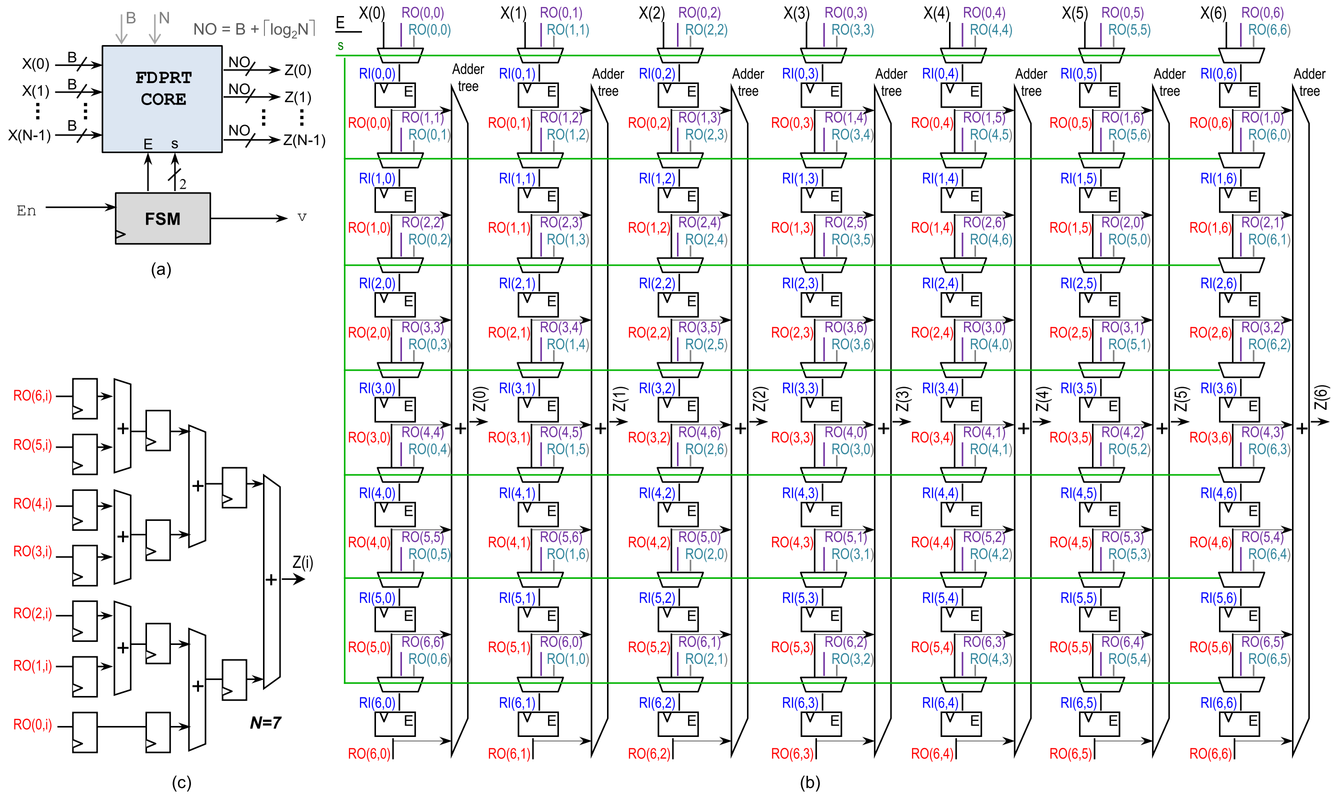


Fig. 8. Fast DPRT (FDPRT) hardware. (a) FDPRT core and finite state machine (FSM). (b) Structure of the FDPRT core including: pipelined adder trees, registers, multiplexers (for shifting and fast transposition) for $N = 7$. (c) Pipelined adder tree architecture for $N = 7$.

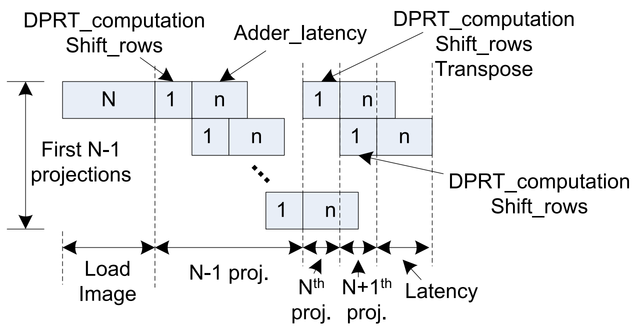

DPRT_computation Shift_rows
Adder_latency
DPRT_computation Shift_rows Transpose
First N-1 projections
DPRT_computation Shift_rows
Load Image
N-1 proj.
Nth proj.
N+1th proj.
Latency

Fig. 9. Running time for fast DPRT (FDPRT). In this diagram, time increases to the right. The DPRT is computed in $N+1$ steps (projections). Each projection takes $1+h$ clock cycles. Here, $n = \lceil \log_2 N \rceil$ represents the addition latency. Pipeline structure: Since we are using fully pipelined adder trees, the computation of subsequent projections can be started after one clock of the previous projection.

```
1: Load $R(m, d)$,
       $0 \le m, d \le N-1$
2: Compute $S$
3: Load $R_N(d)$
4: Compute in parallel $f_0(j)$
5: for $i = 1$ to $N-1$ do
6:     Shift in parallel the last $N-1$ rows
           $CRS_k(k), k = 1, \ldots, N-1$
7:     Compute in parallel $f_i(j)$
8: end for
```

Fig. 10. Algorithm for computing the Inverse Fast Discrete Periodic Radon Transform $f(i, j) = \Re^{-1}(R(m, d))$. We refer to section II-A for the notation.

to process. In other words, we require that:

$$\left\lceil \frac{N}{H} \right\rceil < \left\lceil \frac{N}{H-1} \right\rceil. \quad (11)$$

In this case, using H rows in each strip will result in faster computations since we are processing fewer strips and we are also processing a larger number of rows per strip. The Pareto front is then defined using:

$$\texttt{ParetoFront} = \{H \in S \;\text{ s.t. } H \text{ satisfies eqn (11)}\} \quad (12)$$

where $S = \{2, 3, \ldots, (N-1)/2\}$ denotes the set of possible values for the number of rows. To solve (11) and derive the `ParetoFront` set, we simply plug-in the different values of H and check that (11) is satisfied. Beyond the scalable approach, we note that an optimal architecture for $H = N$ was covered in subsection III-D. We will present the Pareto front in the Results section.

IV. Architecture Implementation

In this section, we will present the architecture implementations for the scalable and fast DPRTs and their inverses. We provide a top-down description of the scalable architecture in section IV-A. Then, for the inverse DPRTs, we show the internal architecture that includes the circular shift registers and the adder trees. Here, we note that the internal architectures for the forward DPRTs are closely related to the architectures for the inverse DPRTs but simpler.

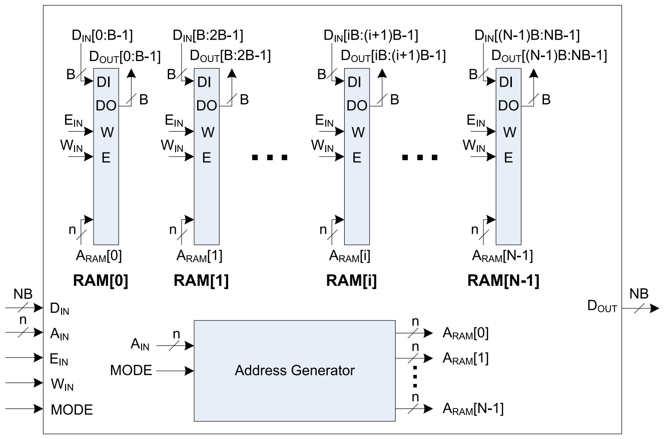


Fig. 11. Memory architecture for parallel read/write. For the parallel load, refer to Fig. 12. The memory allows us to avoid transposition as described in Fig. 13. The memory architecture refers to MEM_IN and MEM_OUT in Fig. 2.

A. Scalable Fast Discrete Periodic Radon Transform (SFDPRT)

In this section, we will analyze and implement the different processes and components that were presented in the top-down diagram of Fig. 2. At the top level, we present block diagrams for the the memory components (MEM_IN and MEM_OUT) in Fig. 11. We will conclude the section with FPGA implementations.

We begin with a brief description of the memory components. Each RAM-block is a standard Random Access Memory with separate address, data read, and data write buses. The *MODE* signal is used to select between row and column access. For row access, the addresses are set to the value stored in $\mathtt{A_{RAM}[0]}$. Column access is only supported for MEM_IN. The addresses for column access are determined using: $\mathtt{A_{RAM}[i]} = \langle \mathtt{A_{RAM}[0]} + \mathtt{i}\rangle_{\mathtt{N}}\,, \mathtt{i} = 1, \ldots, \mathtt{N}-1$.

We summarize the main process of Fig. 3 in four steps:

Step 1: An $N \times N$ image is loaded row-wise in MEM_IN as shown in line 1 of Fig. 12.

Step 2: Image strips are loaded into SFDPRT_core, shifted and written back to MEM_IN as described in Fig. 12). At the end of this step, the image is rearranged so that each diagonal corresponds to an image column. This allows us to get each row of the transposed image in one cycle.

Step 3: Image strips are loaded into the SFDPRT_core and then left-shifted once as described in Fig. 13. For the first N projections, we accumulate the results from partial DPRTs computed for each strip as described in Fig. 14. To compute the accumulated sums, we use an adder array. Also, for pipelined operation, MEM_OUT is implemented as a dual port memory.

Step 4: For the last projection, to avoid transposition, we access the input image in column mode. The rest of the process is the same as for the previous N projections.

The Transform is computed using exact arithmetic using $NO = B + \lceil \log_2 N \rceil$ bits to represent the output where the input uses B-bits per pixel.

```
 1: Load image $f$ into MEM_IN
 2: for $z = 0$ to $K-1$ do
 3:     for $y = 0$ to $H-1$ do
 4:         Move row $(z*H+y)$ of $f$ into SFDPRT_core
              in reverse-order (flipped)
 5:         if $z > 0$ then
 6:             Move the top row from SFDPRT_core to
                  MEM_IN in reverse-order at $((z-1)*H+y)$
 7:         end if
 8:     end for
 9:     Shift in parallel all the $H$ rows into SFDPRT_core
          registers: $CLS(z*H+a)$, $a = 0, \ldots, H-1$.
10: end for
11: for $y = 0$ to $H-1$ do
12:     Move the top row from SFDPRT_core to
          MEM_IN in reverse-order at $((K-1)*H+y)$
13: end for
```

Fig. 12. The implementation of ***Load_shifted_image(f)*** of Fig. 3. The process shifts the input image during the loading process in order to avoid the transposition associated with the last projection. The shifting is performed using the circular left shift registers that are available in SFDPRT_core.

```
1: for $z = 0$ to $H-1$ do
2:     if $M == \text{'}row\_mode\text{'}$ then
3:         Move MEM_IN row $(r \cdot H + z)$, mode $M$
             into SFDPRT_core.
4:     else
5:         Move MEM_IN row $(r \cdot H + z)$, mode $M$
             into SFDPRT_core in reverse-order (flipped).
6:     end if
7: end for
8: Shift in parallel all the $H$ rows:
       $CLS_a(H \cdot r + a)$, $a = 0, \ldots, H-1$
```

Fig. 13. Process for implementing ***Load_strip(r, M)*** of Fig. 3.

```
1: Read accumulated $R_k(d)$ from MEM_OUT
2: if $k = N$ then
3:     Flip $R_k^{'}(d)$
4: end if
5: Add $R_k(d) = R_k(d) + R_k^{'}(d)$
6: Store $R_k(d)$ in MEM_OUT
```

Fig. 14. The implementation of ***Add_partial_result*** of Fig. 3. The process is pipelined where all the steps are executed in a single clock cycle.

B. Inverse Discrete Periodic Transform Implementations

1) iFDPRT: We start by presenting the Inverse Fast Discrete Periodic Radon Transform (iFDPRT) core as shown in Fig. 15. The core generates an $N \times N$ output image based on an $(N+1) \times N$ input array. Fig. 15 shows the array ($N = 7$) where the shift is now to the right. Unlike the FDPRT core, we need to add $R(N, j)$ (an element of the last projection) for each computed direction j. We also need to subtract the sum SR of a row and divide the result by N (2). We do not require transposition of the input array. A total of N directions are generated, where each direction is an N-element vector $F(i)$, $i = 0, \ldots, N-1$.

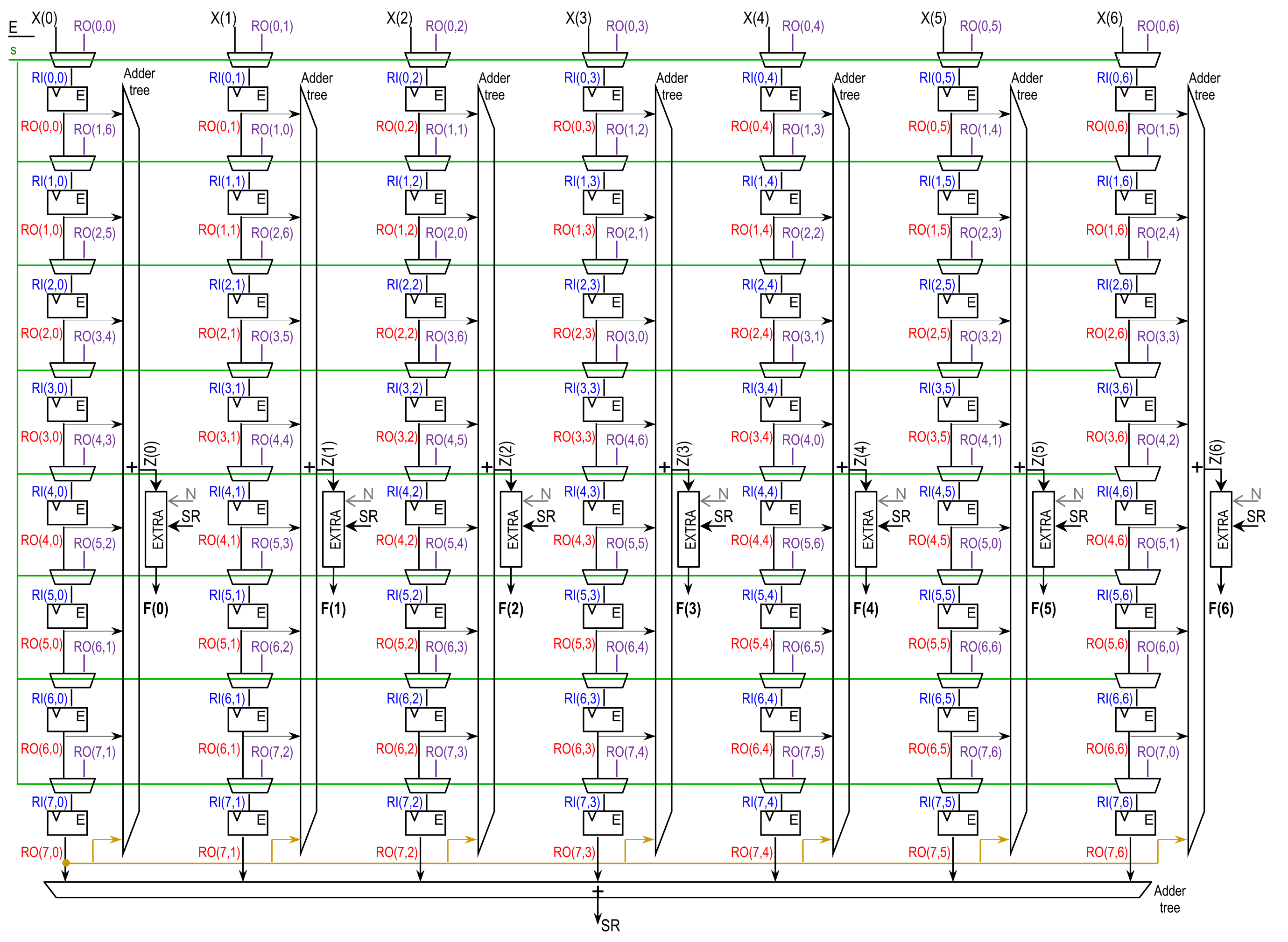


Fig. 15. The fast inverse DPRT (iFDPRT) hardware implementation. The iFDPRT core shows the adder trees, register array, and 2-input MUXes. Here, we note that the $Z(i)$ correspond to the summation term in (2) (also see Fig. 10). We note that the 'extra circuit' is not needed for the forward DPRT. Also, for latency calculations, we note that the 'extra circuit' has a latency of $1 + BO$ cycles.

We next provide a brief overview of the different components. We use 2-input MUXes to support loading and shifting as separate functions. The vertical adder trees generate the $Z(i)$ signals. A new row of $Z(0), Z(1), \ldots, Z(N-1)$ is generated for every cycle. The horizontal adder tree computes SR. We recall that the SR computation is the same for all rows as shown in (3). The latency of the horizontal adder tree is $\lceil \log_2 N \rceil$ cycles. Note that SR is ready when $Z(i)$ is ready, as the latency of the vertical adder trees is $\lceil \log_2 (N+1) \rceil$. The SR value is fed to the 'extra units', where all $Z(i)$'s subtract SR and then divide by N (pipelined array divider [30] with a latency of BO cycles). The term $R(N, j)$ is included by loading the last input row on the last register row, where the shift is one to the left. Note that it is always the same element (the left-most one) that goes to all vertical adders.

We also provide a summary of bitwidth requirements for *perfect reconstruction*. We begin by assuming that the Radon transform coefficients use B'-bits. The number of bits of the vertical adder tree outputs $Z(i)$ are then set to $BO = B' + \lceil \log_2 (N+1) \rceil$. The number of bits of SR need to be $BQ = B' + \lceil \log_2 N \rceil$. Assuming that the input image f is B bits, we only need B bits to reconstruct it and the relationship between B' and B needs to be: $B' = B + \lceil \log_2 N \rceil$ bits. For the subtractor, note that $Z(i) = \sum_{m=0}^{N-1} R\left(m, \langle j - mi \rangle_N\right) + R(N, i)$ and then $Z(i) \geq SR$ since $f(i, j) \geq 0$. Thus, the result of $Z(i) - SR$ will always be positive requiring BO bits. Thus, for perfect reconstruction, the result $F(i)$ needs to be represented using BO bits.

2) `iSFDPRT_core`*:* We next summarize the core for the Inverse Scalable Fast Discrete Periodic Transform core (`iSFDPRT_core`). Fig. 16 shows an instance of this core for $N = 7$ and $H = 4$. The core only generates the partial sums $Z(i)$. We still need to accumulate the partial sums, subtract SR from it and divide by N.

We next provide a summary of the required hardware. For each strip, we need to be able to implement different amounts of right shifting. This is implemented using $(K+1)$-input MUXes. Since N is always prime, we will have at-least one row of the register array that will be unused during computations for the last strip. The unused row is used to load the term $R(N, j)$. The vertical MUXes located on the

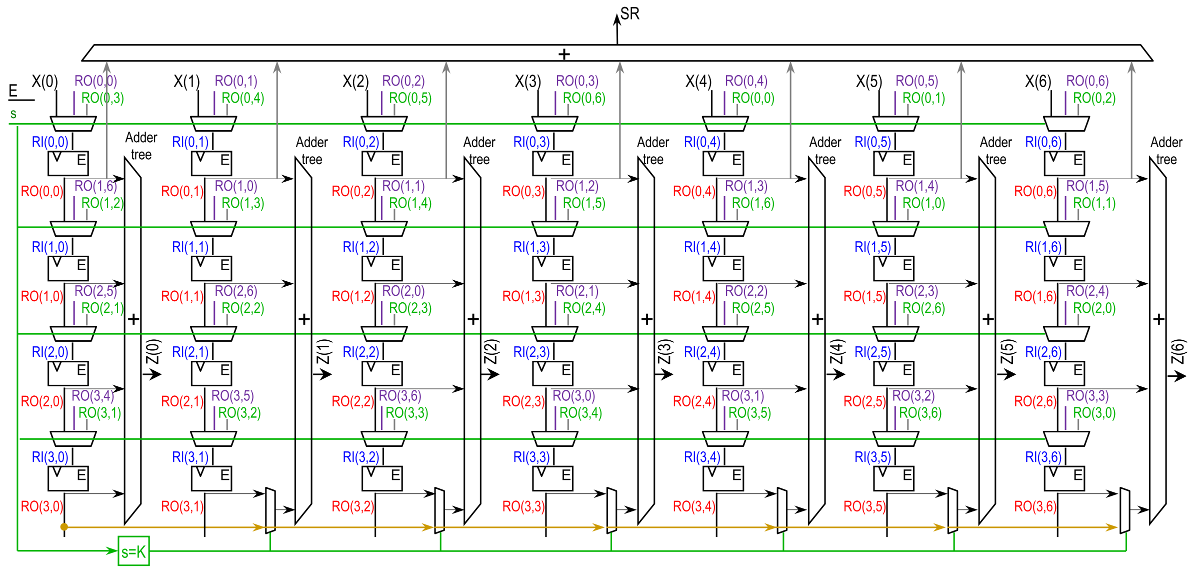


Fig. 16. The inverse scalable DPRT `iSFDPRT_core` architecture for $N = 7$, $H = 4$. Here, we note that the $Z(i)$ correspond to the summation term in (10) (also see Fig. 6).

last valid row of the last strip ensure that the term $R(N, j)$ is considered only when the last strip is being processed. Here, for the last row of the last strip, we require the shift to be one to the left. Also, the remaining unused rows are fed with zeros.

3) iSFDPRT: Recall that we presented the entire system in section III-C. Beyond the `iSFDPRT_core`, we have the input and output memories, an array of adders and divisors, and ancillary logic. Here, we do not need to use the diagonal mode of the memories, flip the data, or rearrange the input memory. The basic process consists of loading each strip, processing it on the `iSFDPRT_core`, accumulating it to the previous result, and storing it in the output memory. For the last strip, we accumulate the result, but we also need to subtract SR and divide by N.

V. Results and Discussion

A. Results

We next provide comprehensive results for both the scalable and the fast DPRTs and their inverses. We also compare the proposed approaches to previously published methods.

We first present results as a function of image size. We summarize running times (in terms of the number of cycles) for the forward and inverse DPRT in Tables I and II respectively. For the forward DPRT, we compare against hardware implementations given by the serial implementation in [19] and the systolic implementation in [20]. For the inverse DPRT, the computation times are similar. However, there are no exact values to compare against. We also show comparative running times for $2 < N < 256$ for $B = 8$ bits per pixel in Fig. 17,

We provide a summary of the computational resources in Table III. We also provide detailed resource functions $\mathtt{A_{ff}}$, $\mathtt{A_{FA}}$, and $\mathtt{A_{mux}}$ usage for the 8-bit 251×251 images in Fig. 18. In Fig. 18, we show resources as a function of the number of rows (H) stored in each image strip. For $N = 251$ and $B = 8$, we also show the required number of RAM resources and the total number of MUXes in Table IV. For comparing performance as a function of resources, we present the required number of cycles as a function of flip-flops in Fig. 19, and as a function of 1-bit additions in Fig. 20.

TABLE I
Total Number of Clock Cycles for Computing the DPRT. In All Cases, the Image Is of Size $N \times N$, and $H = 2, \ldots, N$ Is the Scaling Factor for the SFDPRT

Method	**Clock cycles**
Serial [15],[19]	$N^3 + 2N^2 + N$
Systolic [15],[20]	$N^2 + N + 1$
Proposed Approaches:	
- SFDPRT	$\lceil N/H \rceil (N + 3H + 3) + N + \lceil \log_2 H \rceil + 1$
- SFDPRT ($H = 2$) lowest resource use	$\lceil N/2 \rceil (N + 9) + N + 2$
- SFDPRT ($H = N$) fastest running time	$5N + \lceil \log_2 N \rceil + 4$
- FDPRT	$2N + \lceil \log_2 N \rceil + 1$

TABLE II
Total Number of Clock Cycles for Computing the iDPRT. Here, the Image Size Is $N \times N$. We Use B Bits per Pixel, and $H = 2, \ldots, N$ Is the Scaling Factor of the iSFDPRT. Add N Clock Cycles in the Scalable Version if MEM_IN Is Used

Our work	Clock cycles
iSFDPRT	$\lceil N/H \rceil (N + H) + 2 \lceil \log_2 N \rceil + \lceil \log_2 H \rceil + B + 3$
iSFDPRT ($H = 2$) lowest resource usage	$\lceil N/2 \rceil (N + 2) + 2 \lceil \log_2 N \rceil + B + 4$
iSFDPRT ($H = N$) fastest running time	$2N + 3 \lceil \log_2 N \rceil + B + 3$
iFDPRT	$2N + 3 \lceil \log_2 N \rceil + B + 2$

TABLE III

RESOURCE USAGE FOR DIFFERENT DPRT AND INVERSE DPRT IMPLEMENTATIONS. HERE, WE HAVE AN IMAGE SIZE OF $N \times N$, B BITS PER PIXEL, $n = \lceil \log_2 N \rceil$, $h = \lceil \log_2 H \rceil$, $K = \lceil N/H \rceil$, AND $H = 2, \ldots, N$. FOR THE ADDER TREES, WE DEFINE $\mathtt{A_{ff}}$ TO BE THE NUMBER OF REQUIRED FLIP-FLOPS, AND $\mathtt{A_{FA}}$ TO BE THE NUMBER OF 1-bit ADDITIONS. FOR THE REGISTER ARRAY, WE DEFINE $\mathtt{A_{mux}}$ TO BE THE NUMBER OF 2-TO-1 MUXes. $\mathtt{A_{ff}}$, $\mathtt{A_{FA}}$, AND $\mathtt{A_{mux}}$ GROW LINEARLY WITH RESPECT TO N AND CAN BE COMPUTED USING THE ALGORITHM GIVEN IN THE APPENDIX (FIG. 22). FOR THE INVERSE DPRT, WE NOTE THAT EACH DIVIDER IS IMPLEMENTED USING $3(B+2n)^2$ FLIP-FLOPS, $(B+2n)^2$ 1-bit ADDITIONS, AND $(B+2n)^2$ 2-TO-1 MUXes. HERE, WE USE THE TERM "1-bit ADDITIONS" TO REFER TO THE NUMBER OF EQUIVALENT 1-bit FULL ADDERS

	Resources			
	Register array (in bits)	Adder trees: Number of flip-flops	Adder trees: 1-bit additions	Others: Dividers ($B+2n$ bits) or RAM(in bits), 2-to-1 MUXes
Serial [15],[19]	$N(B+n)$	$3B+2n$	$(B+n)$	RAM: N^2B
Systolic [15],[20]	$N(N+1)n$	$(N+1)(3B+2n)$	$(N+1)(B+n)$	RAM: $N(N+1)(B+n)$
SFDPRT	NHB	$N\mathtt{A_{ff}}(H,B)$	$N\mathtt{A_{FA}}(H,B)$ $+N(B+n)$	RAM: $N^2B+N(N+1)(B+n)$ MUX: $NH\mathtt{A_{mux}}(K+1,B)$
SFDPRT ($H=2$) lowest resource usage	$2NB$	$N(B+1)$	NB $+N(B+n)$	RAM: $N^2B+N(N+1)(B+n)$ MUX: $2N\mathtt{A_{mux}}(\lceil N/2\rceil+1,B)$
SFDPRT ($H=N$) fastest running time	N^2B	$N\mathtt{A_{ff}}(N,B)$	$N\mathtt{A_{FA}}(N,B)$ $+N(B+n)$	RAM: $N^2B+N(N+1)(B+n)$ MUX: N^2B
FDPRT	N^2B	$N\mathtt{A_{ff}}(N,B)$	$N\mathtt{A_{FA}}(N,B)$	MUX: $2N^2B$
iSFDPRT	$NH(B+n)$	$(N+1)\mathtt{A_{ff}}(H,B+n)$ $+3N(B+2n)$	$(N+1)\mathtt{A_{FA}}(H,B+n)$ $+2N(B+2n)$	RAM: $N^2(B+2n)$, Dividers: N MUX: $NH\mathtt{A_{mux}}(K+1,B+n)$
iSFDPRT ($H=2$) lowest resource usage	$2N(B+n)$	$(N+1)(B+n+1)$ $+3N(B+2n)$	$(N+1)(B+n)$ $+2N(B+2n)$	RAM: $N^2(B+2n)$, Dividers: N MUX: $2N\mathtt{A_{mux}}(\lceil N/2\rceil+1,B+n)$
iSFDPRT ($H=N$) fastest running time	$N^2(B+n)$	$(N+1)\mathtt{A_{ff}}(N,B+n)$ $+3N(B+2n)$	$(N+1)\mathtt{A_{FA}}(N,B+n)$ $+2N(B+2n)$	RAM: $N^2(B+2n)$, Dividers: N MUX: $N^2(B+n)$
iFDPRT	$N^2(B+n)$	$(N+1)\mathtt{A_{ff}}(N,B+n)$ $+N(B+2n)$	$(N+1)\mathtt{A_{FA}}(N,B+n)$ $+N(B+2n)$	Dividers: N MUX: $N^2(B+n)$

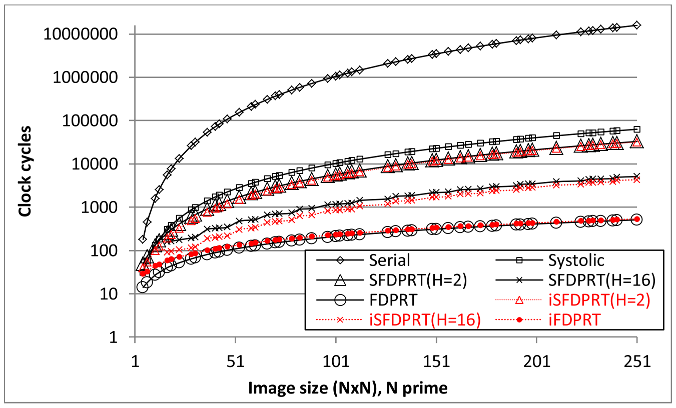


Fig. 17. Comparative running times for the proposed approach versus competitive methods. We report running times in clock cycles for: (i) the serial implementation of [19], (ii) the systolic [20], and (iii) the FPGA implementation of the SFDPRT for $H = 2$ and 16. The measured running times are in agreement with Tables I and II.

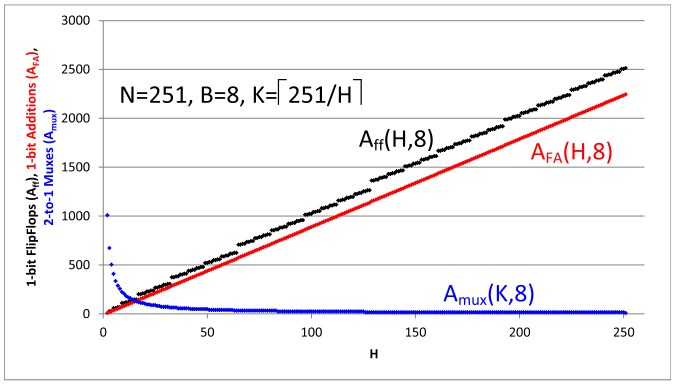


Fig. 18. Resource functions: (i) number of adder tree flip-flops $\mathtt{A_{ff}}(.)$, (ii) number of 1-bit additions $\mathtt{A_{fa}}(.)$, and (iii) number 2-to-1 multiplexers $\mathtt{A_{mux}}(.)$ for $N = 251$, B=8. Refer to Table III for definitions.

We present the required number of slices for a Virtex-6 implementation in Fig. 21. As N increases, we observe linear growth in the number of slices as expected from our analysis in Table III. On the other hand, for smaller values of N, we have quadratic growth. The trends are due to the optimizations performed by the Xilinx synthesizer. Overall, since Virtex-6 devices use 6-input LUTS, implementations that utilize all 6 inputs provide better resource optimization than implementations that use fewer inputs. For the entire system, we have clock frequencies of 100 MHz for the Xilinx 6-series and 200 MHz for the Xilinx 7-series (Virtex-7, Artix-7, Kintex-7).

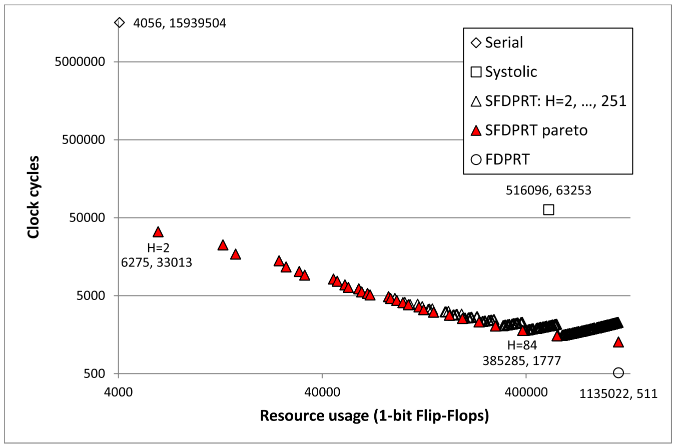


Fig. 19. Comparative plot for the different implementations based on the number of cycles and the number of flip-flops only. Refer to Fig. 20 for a comparative plot for the different implementations based on the number of cycles and the number of 1-bit additions. Also, refer to Table IV for a summary of RAM and multiplexer resources. The plot shows the Pareto front for the proposed SFDPRT for $H = 2, \ldots, 251$, for an image of size 251×251. The Pareto front is defined in terms of running time (in clock cycles) and the number of flip-flops used. For comparison, we show the serial implementation from [19], and the systolic implementation [20]. The fastest implementation is due to the FDPRT that is also shown.

TABLE IV

TOTAL NUMBER OF RESOURCES FOR RAM (IN 1-bit CELLS) AND MUXes (2-TO-1 muxes). THE RESOURCES ARE SHOWN FOR $N = 251$. EXCEPT FOR THE MUXes FOR THE SFDPRT, THE VALUES REFER TO ANY H. THE NUMBER OF MUXes FOR THE SFDPRT REFER TO VALUES OF H THAT LIE ON THE PARETO FRONT*

Method	RAM	MUXes
Serial [15],[19]	504, 008	Unknown
Systolic [15],[20]	1, 012, 032	Unknown
SFDPRT	1, 516, 040	506, 016*
FDPRT	0	1, 008, 016

We also provide a summary of our results for the inverse DPRT. For the fast version (iFDPRT) running time and resources, we refer back to Figs. 17 and 21. For the number of input bits, we recall that $B' = B + \lceil \log_2 N \rceil$. Thus, overall, the iFDPRT implementations require more resources and slightly more computational times. Similar comments apply for the scalable, inverse DPRTs (iSFDPRT) shown in Figs. 17 and 21.

B. Discussion

Overall, the proposed approach results in the fastest running times. Even in the slowest cases, our running times are significantly better than any previous implementation. Scalable DPRT computation has also been demonstrated where the required number of cycles can be reduced when more resources are available. Significantly faster DPRT computation is possible for fixed size transforms when the architecture can be implemented using available resources. Furthermore, these results have been extended for the inverse DPRT. However, in some cases, the better running times come at a cost of increased resources. Thus, we also need to discuss how our running times depend on the number of required resources.

For an $N \times N$ image (N prime), the proposed approaches can compute the DPRT in significantly less time than N^2 cycles.

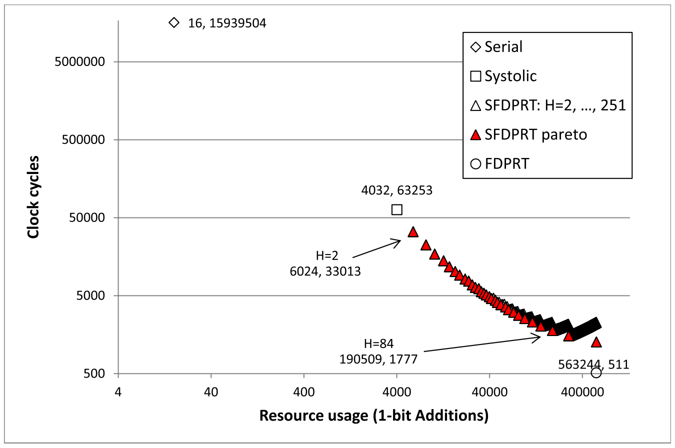


Fig. 20. Comparative plot for the different implementations based on the number of cycles and the number of one-bit additions only (or equivalent 1 bit full adders). Refer to Fig. 19 for a similar comparison based on the number of flip-flops. Pareto front for the proposed SFDPRT for $H = 2, \ldots, 251$, for an image of size 251×251. The Pareto front is defined in terms of running time (in clock cycles) and the number of 1-bit additions. For comparison, we show the serial implementation from [19], and the systolic implementation [20]. The fastest implementation is due to the FDPRT.

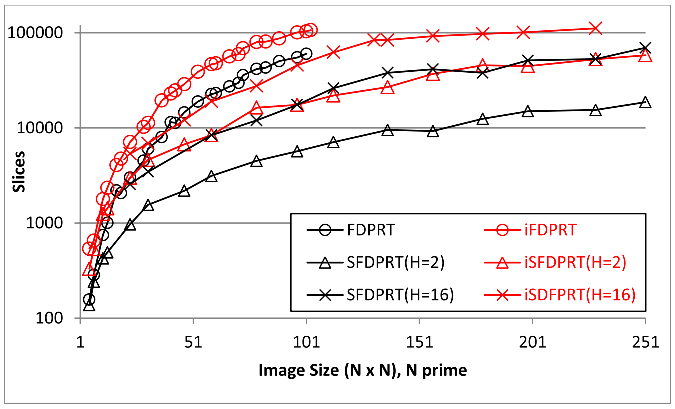


Fig. 21. FPGA slices for a Virtex-6 implementation for both the forward and inverse DPRTs for $H = 2, 16$, N prime and $2 \leq N \leq 251$.

The fastest architectures (FDPRT and iFDPRT) compute the forward DPRT and inverse DPRT in just $2N + \lceil \log_2 N \rceil + 1$ and $2N + 3\lceil \log_2 N \rceil + B + 2$ cycles respectively (where B is the number of bits used to represent each input pixel). When resources are available, the scalable approach can also compute the DPRT in a number of cycles that is linear in N. In the fastest case, the scalable DPRT requires $2N + \lceil \log_2 N \rceil + 1$ clock cycles. However, when very limited resources are available, the number of required clock cycles increases to $\lceil N/2 \rceil (N + 9) + N + 2$ for the case where we only have two image rows per strip $H = 2$.

Based on Fig. 19, we compare the number of cycles as a function of the required number of flip-flops. From the Figure, we note that systolic implementation requires 516, 096 flip flops to compute the DPRT for a 251×251 image in 63, 253 clock cycles (square dot in Fig. 19). In comparison, with 25% less resources for $H = 84$, we have that the scalable DPRT is computed 36 times faster than the systolic implementation. On the other hand, for the serial implementation, we note that the proposed scalable DPRT approaches are much faster but require more resources. The fast DPRT implementation requires only 511 cycles that is vastly superior to any other approach.

Based on Fig. 20, we also compare the number of cycles as a function of the number of 1-bit additions. As expected, the serial implementation requires a single 16-bit adder. However, the serial implementation is very slow compared to all other implementations. The systolic implementation requires only 4,032 1-bit additions that is close to the two-row per strip ($H = 2$) implementation of the scalable DPRT. However, in all cases, the systolic implementation is significantly slower than all of the proposed implementations. Essentially, the scalable approach improves its performance while requiring more 1-bit additions for larger values of H.

As detailed in section III-E, we are only interested in Pareto-optimal implementations. Here, the Pareto-optimal cases represent scalable implementations that always improve performance by using more resources. The collection of all of the Pareto-optimal implementations form the Pareto-front and are shown in Fig. 19 and Fig. 20.

The proposed system can also be expanded for use in FPGA co-processor systems where the FPGA card communicates with the CPU using a PCI express interface. Clearly, the advantage of using the proposed architecture increases with N since the image transfer overhead will not be significant for larger N. To understand the limits, assume a PCI express 3.x bandwidth of about 16 GB/s and a general-purpose microprocessor that achieves the maximum performance of 10 Giga-flops using 4 cores at 2.5 GHz. In terms of CPU memory accesses, we assume a 32GB/second bandwidth for DDR3 memory. Furthermore, suppose that we are interested in computing the DPRT of a 251×251 image. In this case, image I/O requires about 3.67 micro-seconds per image transfer from the DDR3 to the FPGA card. The DPRT requires $N^2 * (N + 1) \approx 15.88$ mega floating point operations for the additions. The additions can be performed in 1.479 milli-seconds (1479 micro-seconds) on the CPU. In addition, the CPU implementation will need $2N^2$ DDR3 memory accesses for implementing the transposition and retrieving the matrix in shifted form. However, assuming that these memory accesses are implemented effectively using DDR3 memory, that only requires 3.67 micro-seconds. Hence, the CPU computation will be dominated by the additions. On the other hand, DPRT computation on an FPGA operating at just 100 MHz for older devices (at half the 200 MHz of more modern FPGA devices), will only require 2*251+9 cycles in about 5.11 micro-seconds. Thus, the speedup factor is above $(3.67 + 1479)/(3.67 + 5.11) \approx 169$.

We also consider comparisons of the DPRT to previous implementations of the Hough transform. Here, we note that the Hough transform can be used to detect lines by adding up edge pixels along different directions. In this application, the Hough transform is computed using the discrete Radon transform (DRT). As noted earlier, prior to adding up values along different directions, typical implementations of the DRT require interpolation. Furthermore, the DRT is defined over the original image as opposed to its periodic extension as required by the DPRT. Despite these limitations of our comparisons, we note that the proposed DPRT implementations are substantially faster than the fastest Hough transform implementations. For example, the authors of [31] report on an FPGA implementation, operating at 200 MHz, that requires 2.07-3.16 ms for detecting lines over 180 orientations in 512×512 images. Similar comments apply to related to continuous-space extension of the Radon transform (e.g., generalized, hyperbolic, parabolic, etc.).

```
1:  procedure Tree_Resources($X, B$)
2:      $h = \lceil \log_2 X \rceil$
3:      $\mathtt{A_{ff}} = \mathtt{A_{FA}} = \mathtt{A_{mux}} = 0$
4:      $a = X$
5:      for $z$ = 1 to $h$ do
6:          $r = \langle a \rangle_2$
7:          $a = \lfloor a/2 \rfloor$
8:          $\mathtt{A_{FA}} = \mathtt{A_{FA}} + a \cdot (B + z - 1)$
9:          $\mathtt{A_{mux}} = \mathtt{A_{mux}} + a \cdot B$
10:         $a = a + r$
11:         $\mathtt{A_{ff}} = \mathtt{A_{ff}} + a \cdot (B + z)$
12:     end for
13:     return $\mathtt{A_{FA}}, \mathtt{A_{ff}}, \mathtt{A_{mux}}$
14: end procedure
```

Fig. 22. Required tree resources as a function of the number of strip rows or number of blocks (X), and the number of bits per pixel (B). Refer to Table III for definitions of $\mathtt{A_{ff}}, \mathtt{A_{FA}}, \mathtt{A_{mux}}$. For $\mathtt{A_{ff}}$, the resources do not include the input registers, but do include the output registers since they are implemented in SFDPRT_core and iSFDPRT_core.

VI. Conclusions and Future Work

The manuscript summarized the development of fast and scalable methods for computing the DPRT and its inverse. Overall, the proposed methods provide much faster computation of the DPRT. Furthermore, the scalable DPRT methods provide fast execution times that can be implemented within available resources. In addition, we present fast DPRT methods that provided the fastest execution times among all possible approaches. For an $N \times N$ image, the fastest DPRT implementations require a number of cycles that grows linearly with N. Furthermore, in terms of resources, the proposed architectures only require fixed point additions and shift registers.

Currently, we are working on the application of the DPRT for computing fast convolutions. As with the current manuscript, our focus will be to extend the current DPRT architecture so as to support the multiplication of the DPRT of the input image with the impulse response of a larger filter, and then take the inverse. Beyond the FPGA implementation, we are also developing GPU implementations. Future work will also focus on the development of fast methods for computing 2D Discrete Fourier Transforms (DFTs).

Appendix

See Fig. 22.

References

[1] A. K. Jain, *Fundamentals of Digital Image Processing*. Upper Saddle River, NJ, USA: Prentice-Hall, 1989.

[2] S. R. Deans, *The Radon Transform and Some of its Applications* (Dover Books on Mathematics). New York, NY, USA: Dover, 2007.

[3] J.-L. Starck, E. J. Candes, and D. L. Donoho, "The curvelet transform for image denoising," *IEEE Trans. Image Process.*, vol. 11, no. 6, pp. 670–684, Jun. 2002.
[4] D. P. K. Lun, T. C. L. Chan, T.-C. Hsung, D. Feng, and Y.-H. Chan, "Efficient blind image restoration using discrete periodic Radon transform," *IEEE Trans. Image Process.*, vol. 13, no. 2, pp. 188–200, Feb. 2004.
[5] K. Jafari-Khouzani and H. Soltanian-Zadeh, "Radon transform orientation estimation for rotation invariant texture analysis," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 27, no. 6, pp. 1004–1008, Jun. 2005.
[6] N. Aggarwal and W. C. Karl, "Line detection in images through regularized Hough transform," *IEEE Trans. Image Process.*, vol. 15, no. 3, pp. 582–591, Mar. 2006.
[7] A. Kingston and I. Svalbe, "Geometric shape effects in redundant keys used to encrypt data transformed by finite discrete radon projections," in *Proc. Digit. Image Comput., Techn. Appl. (DICTA)*, Dec. 2005, p. 16.
[8] N. Normand, I. Svalbe, B. Parrein, and A. Kingston, "Erasure coding with the finite radon transform," in *Proc. IEEE Wireless Commun. Netw. Conf. (WCNC)*, Apr. 2010, pp. 1–6.
[9] B. Parrein, N. Normand, M. Ghareeb, G. D'Ippolito, and F. Battisti, "Finite Radon coding for content delivery over hybrid client-server and P2P architecture," in *Proc. 5th Int. Symp. Commun. Control Signal Process. (ISCCSP)*, May 2012, pp. 1–4.
[10] G.-W. Ou, D. P.-K. Lun, and B. W.-K. Ling, "Compressive sensing of images based on discrete periodic radon transform," *Electron. Lett.*, vol. 50, no. 8, pp. 591–593, Apr. 2014.
[11] G. Beylkin, "Discrete radon transform," *IEEE Trans. Acoust., Speech, Signal Process.*, vol. 35, no. 2, pp. 162–172, Feb. 1987.
[12] B. T. Kelley and V. K. Madisetti, "The fast discrete Radon transform. I. Theory," *IEEE Trans. Image Process.*, vol. 2, no. 3, pp. 382–400, Jul. 1993.
[13] V. V. Vlček, "Computation of inverse radon transform on graphic cards," *Int. J. Signal Process.*, vol. 1, no. 1, pp. 1–12, 2004.
[14] A. M. Grigoryan, "Comments on 'the discrete periodic radon transform,'" *IEEE Trans. Signal Process.*, vol. 58, no. 11, pp. 5962–5963, Nov. 2010.
[15] F. Matúš and J. Flusser, "Image representation via a finite Radon transform," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 15, no. 10, pp. 996–1006, Oct. 1993.
[16] T. Hsung, D. P. K. Lun, and W.-C. Siu, "The discrete periodic Radon transform," *IEEE Trans. Signal Process.*, vol. 44, no. 10, pp. 2651–2657, Oct. 1996.
[17] I. Gertner, "A new efficient algorithm to compute the two-dimensional discrete Fourier transform," *IEEE Trans. Acoust., Speech, Signal Process.*, vol. 36, no. 7, pp. 1036–1050, Jul. 1988.
[18] A. Ahmad, A. Amira, H. Rabah, and Y. Berviller, "Medical image denoising on field programmable gate array using finite Radon transform," *IET Signal Process.*, vol. 6, no. 9, pp. 862–870, Dec. 2012.
[19] S. Chandrasekaran and A. Amira, "High speed/low power architectures for the finite radon transform," in *Proc. Int. Conf. Field Program. Logic Appl.*, Aug. 2005, pp. 450–455.
[20] S. Chandrasekaran, A. Amira, S. Minghua, and A. Bermak, "An efficient VLSI architecture and FPGA implementation of the finite ridgelet transform," *J. Real-Time Image Process.*, vol. 3, no. 3, pp. 183–193, Sep. 2008.
[21] A. Kingston and I. Svalbe, "Projective transforms on periodic discrete image arrays," in *Proc. Adv. Imag. Electron Phys.*, vol. 139. 2006, p. 76.
[22] M. S. Pattichis, "Novel algorithms for the accurate, efficient, and parallel computation of multidimensional, regional discrete Fourier transforms," in *Proc. 10th Medit. Electrotech. Conf. (MELECON)*, vol. 2. May 2000, pp. 530–533.
[23] M. S. Pattichis and R. Zhou, "A novel directional approach for the scalable, accurate and efficient computation of two-dimensional discrete Fourier transforms," Dept. Elect. Comput. Eng., Univ. New Mexico, Albuquerque, NM, USA, Tech. Rep. AHPCC2000-019, 2000.
[24] M. S. Pattichis, R. Zhou, and B. Raman, "New algorithms for computing directional discrete Fourier transforms," in *Proc. ICIP*, vol. 3. Oct. 2001, pp. 322–325.
[25] G. H. Hardy and E. M. Wright, *An Introduction to the Theory of Numbers*. London, U.K.: Oxford Univ. Press, 1979.
[26] C. Carranza, D. Llamocca, and M. Pattichis, "The fast discrete periodic radon transform for prime sized images: Algorithm, architecture, and VLSI/FPGA implementation," in *Proc. IEEE Southwest Symp. Image Anal. Interpretation (SSIAI)*, Apr. 2014, pp. 169–172.
[27] C. Carranza, D. Llamocca, and M. Pattichis, "A scalable architecture for implementing the fast discrete periodic radon transform for prime sized images," in *Proc. IEEE Int. Conf. Image Process. (ICIP)*, Paris, France, Oct. 2014, pp. 1208–1212.
[28] S. Boyd and L. Vandenberghe, *Convex Optimization*. Cambridge, U.K.: Cambridge Univ. Press, 2004.
[29] D. Llamocca and M. Pattichis, "A dynamically reconfigurable pixel processor system based on power/energy-performance-accuracy optimization," *IEEE Trans. Circuits Syst. Video Technol.*, vol.'23, no. 3, pp. 488–502, Mar. 2013. [Online]. Available: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6252023
[30] B. Parhami, *Computer Arithmetic: Algorithms and Hardware Designs*, 2nd ed. New York, NY, USA: Oxford Univ. Press, 2009.
[31] Z.-H. Chen, A. W. Y. Su, and M.-T. Sun, "Resource-efficient FPGA architecture and implementation of Hough transform," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 20, no. 8, pp. 1419–1428, Aug. 2012.

Cesar Carranza received the B.Sc. degree in electrical engineering from Pontificia Universidad Católica del Perú, in 1994, the M.Sc. degree in computer science from the Centro de Investigación Científica y de Educación Superior de Ensenada, in 2010, and the M.Sc. degree in computer engineering from the University of New Mexico, Albuquerque, in 2012, where he is currently pursuing the Ph.D. degree in computer engineering. He is currently an Assistant Professor with Pontificia Universidad Católica del Perú. His current research interests include parallel algorithms for image processing, high-performance hardware integration, and parallel computing.

Daniel Llamocca received the B.Sc. degree in electrical engineering from Pontificia Universidad Católica del Perú, in 2002, and the M.Sc. degree in electrical engineering and the Ph.D. degree in computer engineering from the University of New Mexico at Albuquerque, in 2008 and 2012, respectively.

He is currently an Assistant Professor with Oakland University. His research deals with run-time automatic adaptation of hardware resources to time-varying constraints with the purpose of delivering the best hardware solution at any time. His current research interests include reconfigurable computer architectures for signal, image, and video processing, high-performance architectures for computer arithmetic, communication, and embedded interfaces, embedded system design, and run-time partial reconfiguration techniques on field-programmable gate arrays.

Marios Pattichis (M'99–SM'06) received the B.Sc. (Hons.) degree in computer sciences and the B.A. (Hons.) degree in mathematics, the M.S. degree in electrical engineering, and the Ph.D. degree in computer engineering from The University of Texas at Austin, in 1991, 1993, and 1998, respectively. He was a Founding Co-PI of COSMIAC at the University of New Mexico (UNM), Albuquerque, where he is currently a Professor of the Department of Electrical and Computer Engineering and the Director of the Image and Video Processing and Communications Laboratory. His current research interests include digital image, video processing, communications, dynamically reconfigurable computer architectures, and biomedical and space image-processing applications.

Dr. Pattichis was a recipient of the 2004 Electrical and Computer Engineering Distinguished Teaching Award at UNM. For his development of the digital logic design laboratories at UNM, he was recognized by Xilinx Corporation in 2003 and by the UNM School of Engineering's Harrison Faculty Excellent Award in 2006. He was the General Chair of the 2008 IEEE Southwest Symposium on Image Analysis and Interpretation. He is currently a Senior Associate Editor of the *IEEE Signal Processing Letters*. He has served as an Associate Editor of the IEEE TRANSACTIONS ON IMAGE PROCESSING and the IEEE TRANSACTIONS ON INDUSTRIAL INFORMATICS, and has also served as a Guest Associate Editor of the IEEE TRANSACTIONS ON INFORMATION TECHNOLOGY IN BIOMEDICINE.